Adaptive Low-level Storage of Very Large Knowledge Graphs

Urbani, Jacopo; Jacobs, Ceriel

published in
WWW '20
2020

DOI (link to publisher)
10.1145/3366423.3380246

document version
Publisher's PDF, also known as Version of record

Adaptive Low-level Storage of Very Large Knowledge Graphs

Jacopo Urbani
jacopo@cs.vu.nl
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Ceriel Jacobs
ceriel@cs.vu.nl
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

ABSTRACT

The increasing availability and usage of Knowledge Graphs (KGs) on the Web calls for scalable and general-purpose solutions to store this type of data structures. We propose Trident, a novel storage architecture for very large KGs on centralized systems. Trident uses several interlinked data structures to provide fast access to nodes and edges, with the physical storage changing depending on the topology of the graph to reduce the memory footprint. In contrast to single architectures designed for single tasks, our approach offers an interface with few low-level and general-purpose primitives that can be used to implement tasks like SPARQL query answering, reasoning, or graph analytics. Our experiments show that Trident can handle graphs with 10^{11} edges using inexpensive hardware, delivering competitive performance on multiple workloads.

ACM Reference Format:
Jacopo Urbani and Ceriel Jacobs. 2020. Adaptive Low-level Storage of Very Large Knowledge Graphs. In *Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3366423.3380246

1 INTRODUCTION

Motivation. Currently, a large wealth of knowledge is published on the Web in the form of interlinked Knowledge Graphs (KGs). These KGs cover different fields (e.g., biomedicine [17, 57], encyclopedic or commonsense knowledge [77, 84], etc.), and are actively used to enhance tasks such as entity recognition [72], query answering [88], or more in general Web search [14, 28, 29].

As the size of KGs keeps growing and their usefulness expands to new scenarios, applications increasingly need to access large KGs for different purposes. For instance, a search engine might need to query a KG using SPARQL [36], enrich the results using embeddings of the graph [56], and then compute some centrality metrics for ranking the answers [41, 78]. In such cases, the storage engine must not only efficiently handle large KGs, but also allow the execution of multiple types of computation so that the same KG does not have to be loaded in multiple systems.

Problem. In this paper, we focus on providing an efficient, scalable, and general-purpose storage solution for large KGs on centralized architectures. A large amount of recent research has focused on distributed architectures [18, 25, 26, 31, 32, 47, 66, 71], because they offer many cores and a large storage space. However, these benefits come at the price of higher communication cost and increased system complexity [63]. Moreover, sometimes distributed solutions cannot be used either due to financial or privacy-related constraints.

Centralized architectures, in contrast, do not have network costs, are commonly affordable, and provide enough resources to load all-but-the-largest graphs. Some centralized storage engines have demonstrated that they can handle large graphs, but they focus primarily on supporting one particular type of workload (e.g., Ringo [63] supports graph analytics, RDF engines like Virtuoso [59] or RDFox [52] focus on SPARQL [36]). To the best of our knowledge, we still lack a single storage solution that can handle very large KGs as well as support multiple workloads.

Our approach. In this paper, we fill this gap presenting Trident, a novel storage architecture that can store very large KGs on centralized architectures, support multiple workloads, such as SPARQL querying, reasoning, or graph analytics, and is resource-savvy. Therefore, it meets our goal of combining scalability and general-purpose computation.

We started the development of Trident by studying which are the most frequent access types performed during the execution of tasks like SPARQL answering, reasoning, etc. Some of these access types are *node-centric* (i.e., access subsets of the nodes), while others are *edge-centric* (i.e., access subsets of the edges). From this study, we distilled a small set of low-level primitives that can be used to implement more complex tasks. Then, the research focused on designing an architecture that supports the execution of these primitives as efficiently as possible, resulting in Trident.

At its core, Trident uses a dedicated data structure (a B+Tree or an in-memory array) to support fast access to the nodes, and a series of binary tables to store subsets of the edges. Since there can be many binary tables – possibly billions with the largest KGs – handling them with a relational DBMS can be problematic. To avoid this problem, we introduce a light-weight storage scheme where the tables are serialized on byte streams with only a little overhead per table. In this way, tables can be quickly loaded from the secondary storage without expensive pre-processing and offloaded in case the size of the database exceeds the amount of available RAM.

Another important benefit of our approach is that it allows us to exploit the topology of the graph to reduce its physical storage. To this end, we introduce a novel procedure that analyses each binary table and decides, at loading time, whether the table should be stored either in a row-by-row, column-by-column, or in a cluster-based fashion. In this way, the storage engines effectively *adapts* to the input. Finally, we introduce other dynamic procedures that decide, at loading time, whether some tables can be ignored due to their small sizes or whether the content of some tables can be aggregated to further reduce the space.

Since Trident offers low-level primitives, we built interfaces to several engines (RDF3X [55], VLog [81], SNAP [45]) to evaluate

WWW '20, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7023-3/20/04.
https://doi.org/10.1145/3366423.3380246

the performance of SPARQL query answering, datalog reasoning and graph analytics on various types of graphs. Our comparison against the state-of-the-art shows that our approach can be highly competitive in multiple scenarios.

Contribution. We identified the followings as the main contributions of this paper.

- We propose a new architecture to store very large KGs on a centralized system. In contrast to other engines that store the KG in few data structures (e.g., relational tables), our architecture exhaustively decomposes the storage in many binary tables such that it supports both node- and edge-centric via a small number of primitives;
- Our storage solution adapts to the KG as it uses different layouts to store the binary tables depending on its topology. Moreover, some binary tables are either skipped or aggregated to save further space. The adaptation of the physical storage is, as far as we know, a unique feature which is particularly useful for highly heterogeneous graphs, such as the KGs on the Web;
- We present an evaluation with multiple workloads and the results indicate highly competitive performance while maintaining good scalability. In some of our largest experiments, Trident was able to load and process KGs with up to 10^{11} (100B) edges with hardware that costs less than $5K.

The source code of Trident is freely available with an open source license at https://github.com/karmaresearch/trident, along with links to the datasets and instructions to replicate our experiments. An extended version of this paper is available at [80].

2 PRELIMINARIES

A *graph* $G = (V, E, L, \phi_V, \phi_E)$ is a tuple where V, E, L represent the sets of nodes, edges and labels respectively, ϕ_V is a bijection that maps each node to a label in L, while ϕ_E is a function that maps each edge to a label in L. We assume that there is at most one edge with the same label between any pair of nodes. Throughout, we use the notation $r(s, d)$ to indicate the edge with label r from the node with label s (source) to the node with label d (destination).

We say that the graph is *undirected* if $r(s, d) \in E$ implies that also $r(d, s) \in E$. Otherwise, the graph is *directed*. A graph is *unlabeled* if all edges map to the same label. In this paper, we will mostly focus on labeled directed graphs since undirected or unlabeled graphs are special cases of labeled directed graphs.

In practice, it is inefficient to store the graph using the raw labels as identifiers. The most common strategy, which is the one we also follow, consists of assigning a numerical ID to each label in L, and stores each edge $r(s, d)$ with the tuple $\langle \iota_s, \iota_r, \iota_d \rangle$ where ι_s, ι_r, and ι_d are the IDs associated to s, r, and d respectively.

The numerical IDs allow us to sort the edges and by permuting $\iota_s, \iota_r, \iota_d$ we can define six possible ordering criteria. We use strings of three characters over the alphabet $\{\mathtt{s}, \mathtt{r}, \mathtt{d}\}$ to identify these orderings, e.g., `srd` specifies that the edges are ordered by source, relation, and destination. We denote with $\mathcal{R} = \{\mathtt{srd}, \mathtt{sdr}, \ldots\}$ the collection of six orderings while $\mathcal{R}' = \{\mathtt{s}, \mathtt{r}, \mathtt{d}, \mathtt{sr}, \mathtt{rs}, \mathtt{sd}, \mathtt{ds}, \mathtt{dr}, \mathtt{rd}\}$ specifies all partial orderings. We use the function isprefix to check whether string a is a prefix of b, i.e., $\mathrm{isprefix}(a, b) = [true|false]$ and the operator $-$ to remove all characters of one string from another one (e.g., if $a = \mathtt{srd}$ and $b = \mathtt{sd}$, then $a - b = \mathtt{r}$).

Let $\mathcal{V}$ be a set of variables. A *simple graph pattern* (or *triple pattern*) is an instance of $L \cup \mathcal{V} \times L \cup \mathcal{V} \times L \cup \mathcal{V}$ and we denote it as (X, Y, Z) where $X, Y, Z \in L \cup \mathcal{V}$. A *graph pattern* is a finite set of simple graph patterns. Let $\sigma : \mathcal{V} \rightarrow L$ be a partial function from variables to labels. With a slight abuse of notation, we also use σ as a postfix operator that replaces each occurrence of the variables in σ with the corresponding node. Given the graph G and a simple graph pattern q, the *answers* for q on G correspond to the set $\mathrm{ans}(G, q) = \{r(s, d) \mid r(s, d) \in E \wedge q\sigma = (s, r, d)\}$. Function $\mathrm{bound}(p)$ returns the positions of the labels in the simple graph pattern p left-to-right, i.e., if $p = (X, a, b)$ where $X \in \mathcal{V}$ and $a, b \in L$, then $\mathrm{bound}(p) = \mathtt{rd}$.

A *Knowledge Graph (KG)* is a directed labeled graph where nodes are entities and edges establish semantic relations between them, e.g., $\langle Sadiq\_Khan, majorOf, London \rangle$. Usually, KGs are published on the Web using the RDF data model [40]. In this model, data is represented as a set of *triples* of the form $\langle subject, predicate, object \rangle$ drawn from $(\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{L})$ where $\mathcal{I}, \mathcal{B}, \mathcal{L}$ denote sets of IRIs, blank nodes and literals respectively. Let $\mathcal{T} = \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ be the set of all RDF terms. RDF triples can be trivially seen as a graph where the subjects and objects are the nodes, triples map to edges labeled with their predicate name, and $L = \mathcal{T}$.

3 GRAPH PRIMITIVES

We start our discussion with a description of the low-level primitives that we wish to support. We distilled these primitives considering four types of workloads: *SPARQL [36] query answering*, which is the most popular language for querying KGs; *Rule-based reasoning* [7], which is an important task in the Semantic Web to infer new knowledge from KGs; Algorithms for *graph analytics*, or network analysis, since these are widely applied on KGs either to study characteristics like the graph's topology or degree distribution, or within more complex pipelines; *Statistical relational models* [56], which are effective techniques to make predictions using the KG as prior evidence.

If we take a closer look at the computation performed in these tasks, we can make a first broad distinction between *edge-centric* and *node-centric* operations. The first ones can be defined as operations that retrieve subsets of edges that satisfy some constraints. In contrast, operations of the second type retrieve various data about the nodes, like their degree. Some tasks, like SPARQL query answering, depend more heavily on edge-centric operations while others depend more on node-centric operations (e.g., random walks).

Graph Primitives. Following a RISC-like approach, we identified a small number of low-level primitives that can act as basic building blocks for implementing both node- and edge-centric operations. These primitives are reported in Table 1 and are described below.

$\mathbf{f_1} - \mathbf{f_4}$. These primitives retrieve the numerical IDs associated with labels and vice-versa. The primitives f_1 and f_2 retrieve the labels associated with nodes and edges respectively. The primitives f_3 and f_4 retrieve the labels associated with numerical IDs.

$\mathbf{f_5} - \mathbf{f_{10}}$. Function $\mathrm{edg}_\omega(G, p)$ retrieves the subset of the edges in G that matches the simple graph pattern p and returns it sorted according to ω. Primitives in this group are particularly important

	Name	*Output*
f_1	$\text{lbl}_n(G, n)$	Label of node n (equals to $\phi_V(v)$).
f_2	$\text{lbl}_e(G, e)$	Label of edge e (equals to $\phi_E(e)$).
f_3	$\text{nodid}(G, l)$	ι_l, i.e., the ID of node with label l.
f_4	$\text{edgid}(G, l)$	ι_l, i.e., the ID of edge label l.
f_5	$\text{edg}_{\texttt{srd}}(G, p)$	$\text{ans}(G, p)$ sorted by srd.
f_6	$\text{edg}_{\texttt{sdr}}(G, p)$	$\text{ans}(G, p)$ sorted by sdr.
f_7	$\text{edg}_{\texttt{drs}}(G, p)$	$\text{ans}(G, p)$ sorted by drs.
f_8	$\text{edg}_{\texttt{dsr}}(G, p)$	$\text{ans}(G, p)$ sorted by dsr.
f_9	$\text{edg}_{\texttt{rsd}}(G, p)$	$\text{ans}(G, p)$ sorted by rsd.
f_{10}	$\text{edg}_{\texttt{rds}}(G, p)$	$\text{ans}(G, p)$ sorted by rds.
f_{11}	$\text{grp}_{\texttt{s}}(G, p)$	All s of $\text{ans}(G, p)$.
f_{12}	$\text{grp}_{\texttt{r}}(G, p)$	All r of $\text{ans}(G, p)$.
f_{13}	$\text{grp}_{\texttt{d}}(G, p)$	All d of $\text{ans}(G, p)$.
f_{14}	$\text{grp}_{\{\texttt{sr},\texttt{sd}\}}(G, p)$	Aggr. $(s, r)/(s, d)$ of $\text{ans}(G, p)$.
f_{15}	$\text{grp}_{\{\texttt{rs},\texttt{rd}\}}(G, p)$	Aggr. $(r, s)/(r, d)$ of $\text{ans}(G, p)$.
f_{16}	$\text{grp}_{\{\texttt{ds},\texttt{dr}\}}(G, p)$	Aggr. $(d, s)/(d, r)$ of $\text{ans}(G, p)$.
f_{17}	$\text{count}(f_5 \mid \ldots \mid f_{16})$	Cardinality of $f_5, \ldots, f_{16}$.
f_{18}	$\text{pos}_{\texttt{srd}}(G, p, i)$	i^{th} edge returned by $\text{edg}_{\texttt{srd}}(G, p)$.
f_{19}	$\text{pos}_{\texttt{sdr}}(G, p, i)$	i^{th} edge returned by $\text{edg}_{\texttt{sdr}}(G, p)$.
f_{20}	$\text{pos}_{\texttt{drs}}(G, p, i)$	i^{th} edge returned by $\text{edg}_{\texttt{drs}}(G, p)$.
f_{21}	$\text{pos}_{\texttt{dsr}}(G, p, i)$	i^{th} edge returned by $\text{edg}_{\texttt{dsr}}(G, p)$.
f_{22}	$\text{pos}_{\texttt{rsd}}(G, p, i)$	i^{th} edge returned by $\text{edg}_{\texttt{rsd}}(G, p)$.
f_{23}	$\text{pos}_{\texttt{rds}}(G, p, i)$	i^{th} edge returned by $\text{edg}_{\texttt{rds}}(G, p)$.

Table 1: Graph primitives

for the execution of SPARQL queries since they encode the core operation of retrieving the answers of a SPARQL triple pattern.

$\mathbf{f_{11} - f_{16}}$. This group of primitives returns an aggregated version of the output of $\mathbf{f_5 - f_{10}}$. For instance, $\text{grp}_s(G, p)$ returns the list $\langle (x_1, c_1), \ldots, (x_n, c_n) \rangle$ of all distinct sources in the edges $\text{ans}(G, p)$ with the respective counts of the edges that share them. Let D be a set of edges, $A(x, D) = \{r(x, d) \in D\}$ and $B(D) = \{(s, c) \mid r(s, d) \in A(s, D) \wedge c = |A(s, D)|\}$. Then, $\text{grp}_s(G, p)$ returns the list of all tuples in $B(\text{ans}(G, p))$ sorted by the numerical ID of the first field. The other primitives are defined analogously.

$\mathbf{f_{17}}$. This primitive returns the cardinality of the output of $f_5, \ldots, f_{16}$. This computation is useful in a number of cases: For instance, it can be used to optimize the computation of SPARQL queries by rearranging the join ordering depending on the cardinalities of the triple patterns or to compute the degree of nodes in the graph.

$\mathbf{f_{18} - f_{23}}$. These primitives return the i^{th} edge that would be returned by the corresponding primitives edg_*. In practice, this operation is needed in several graph analytics algorithms or for mini-batching during the training of statistical relational models.

4 ARCHITECTURE

One straightforward way to implement the primitives in Table 1 is to store the KG in many independent data structures that provide optimal access for each function. However, such solution will require a large amount of space and updates will be slow. It is challenging to design a storage engine that uses fewer data structures without excessively compromising the performance.

Moreover, KGs are highly heterogeneous objects where some subgraphs have a completely different topology than others. The storage engine should take advantage of this diversity and potentially store different parts of the KGs in different ways, effectively adapting to its structure. This adaptation lacks in current engines, which treat the KG as a single object to store.

Our architecture addresses these two problems with a compact storage layer that supports the execution of primitives $f_1, \ldots, f_{23}$ with a minimal compromise in terms of performance, and in such a way that the engine can adapt to the KG in input selecting the best strategy to store its parts.

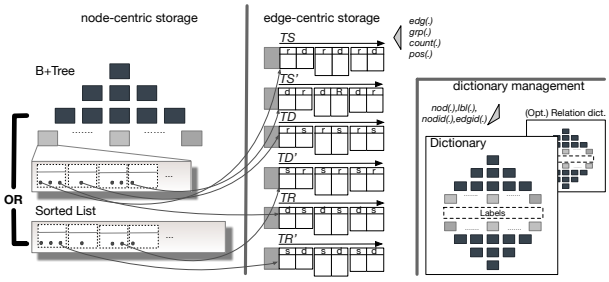


Figure 1: Architectural overview of Trident

Figure 1 gives a graphical view of our approach. It uses a series of interlinked data structures that can be grouped in *three components*. The first one contains data structures for the mappings $ID \Leftrightarrow label$. The second component is called *edge-centric storage* and contains data structures for providing fast access to the edges. The third one is called *node-centric storage* and offers fast access to the nodes. Section 4.1 describes these components in more detail. Section 4.2 discusses how they allow an efficient execution of the primitives, while Section 4.3 focuses on loading and updating the database.

4.1 Architectural Components

Dictionary. We store the labels on a block-based byte stream on disk. We use one B+Tree called $\texttt{DICT}_\iota$ to index the mappings $ID \Rightarrow label$ and another one called $\texttt{DICT}_1$ for $label \Rightarrow ID$. Using B+Trees here is usual, so we will not discuss it further. Note that Trident supports both global ID assignments and independent entity/relationship assignments with an additional index specifically for the relation labels. The first type of assignment is needed for tasks like SPARQL query answering while the second is useful for operations like learning graph embeddings [56].

Edge-centric storage. In order to adapt to the complex and non-uniform topology of current KGs, we do not store all edges in a single data structure, but store subsets of the edges independently. These subsets correspond the edges which share a specific entity/relation. More specifically, let us assume that we must store the graph $G = (V, E, L, \phi_V, \phi_E)$. For each $l \in L$, we consider three types of subsets: $E_s(l) = \{r(l, d) \in E\}$, $E_r(l) = \{l(s, d) \in E\}$, $E_d(l) = \{r(s, l) \in E\}$, i.e., the subsets of edges that have l as source, edge, or destination respectively.

The choice of separating the storage of various subsets allows us to choose the best data structure for a specific subset, but it hinders the execution of *inter-table scans*, i.e., scans where the content of multiple tables must be taken into account. To alleviate this problem, we organize the physical storage in such a way that all edges can still be retrieved by scanning a contiguous memory location.

We proceed as follows: First, we compute $E_s(l)$, $E_r(l)$, $E_d(l)$ for every $l \in L$. Let Ω be the collection of all these sets. For each $E_x(l) \in \Omega$, we construct two sets of tuples, $F_x(l)$ and $G_x(l)$, by extracting the free fields left-to-right and right-to-left respectively. For instance, the set $E_s(l)$ results into the sets $F_s(l) = \{\langle r, d \rangle \mid r(l, d) \in E\}$ and

$G_s(l) = \{\langle d, r\rangle \mid r(l, d) \in E\}$. Since these sets contains pairs of elements, we view them as binary tables. These are grouped into the following six sets:

- $T_s = \{F_s(l) \mid l \in L\}$ and $T'_s = \{G_s(l) \mid l \in L\}$
- $T_r = \{F_r(l) \mid l \in L\}$ and $T'_r = \{G_r(l) \mid l \in L\}$
- $T_d = \{F_d(l) \mid l \in L\}$ and $T'_d = \{G_d(l) \mid l \in L\}$

The content of these six sets is serialized on disk in corresponding byte streams called TS, TS′, TR, TR′, TD, and TD′ respectively (see middle section of Figure 1). The serialization is done by first sorting the binary tables by their defining label IDs, and then serializing each table one-by-one. For instance, if $F_s(l_1), F_s(l_2) \in T_s$, then $F_s(l_1)$ is serialized before $F_s(l_2)$ iff $\iota_{l_1} < \iota_{l_2}$. At the beginning of the byte stream, we store the list of all IDs associated to the tables, pointers to the tables' physical location and instructions to parse them.

Since the binary tables and tuples are serialized on the byte stream with a specific order, we can retrieve all edges sorted with any ordering in $\mathcal{R}$ with a single scan of the corresponding byte stream, using the content stored at the beginning of the stream to decode the binary tables in it. For instance, we can scan TS to retrieve all edges sorted according to srd.

Node-centric storage. In order to provide fast access to the nodes, we map each ID ι_l (i.e., the ID assigned to label l) to a tuple M_l that contains 15 fields:

- the cardinalities $|E_s(l)|$, $|E_r(l)|$, and $|E_d(l)|$;
- Six pointers $p_1, \ldots, p_6$ to the physical storage of $F_s(l)$, $G_s(l)$, $F_r(l)$, $G_r(l)$, $F_d(l)$, and $G_d(l)$;
- Six bytes $m_1, \ldots, m_6$ that contain instructions to read the data structures pointed by $p_1, \ldots, p_6$. These instructions are necessary because the tables are stored in different ways (see Section 5).

We index all M_* tuples by the numerical IDs using a single global data structure called NM (Node Manager), shown on the left side of Figure 1. This data structure is implemented either with an on-disk B+Tree or with an in-memory sorted vector (the choice is done at loading time). The B+Tree is preferable if the engine is used for edge-based computation because the B+Tree does not need to load all nodes in main memory and the nodes are accessed infrequently anyway. In contrast, the sorted vector provides much faster access ($O(1)$ vs. $O(log|L|)$) but it requires that the entire vector is stored in main memory. Thus, it is suitable only if the application accesses the nodes very frequently and there are enough hardware resources.

The way we store the binary tables in six byte streams resembles six-permutation indexing schemes such as proposed in engines like RDF3X [55] or Hexastore [85]. There are, however, two important differences: First, in our approach the edges are stored in multiple independent binary tables rather than a single series of ternary tuples (as, for instance, in RDF3X [55]). This division is important because it allows us to choose different serialization strategies for subgraphs or to avoid storing some tables (Section 5.3). The second difference is that in our case most access patterns go through a *single* B+Tree instead of six different data structures. This allows us to save space and to store additional information about the nodes, e.g., their degree, which is useful, for instance, for traversal algorithms like PageRank, or random walks.

4.2 Primitive Execution

We now discuss how we can implement the primitives in Table 1 with our architecture.

Primitives $\mathbf{f_1}, \ldots, \mathbf{f_4}$ (lbl_* nodid, edgid). These primitives are executed consulting $\mathtt{DICT}_1$ or $\mathtt{DICT}_\iota$. Thus, the following holds.

Proposition 1. *Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of computing $f_1, \ldots, f_4$ is $O(log(|L|))$.*

Primitives $\mathbf{f_5}, \ldots, \mathbf{f_{10}}$ (edg_*). Let $\mathsf{edg}_\omega(G, p)$ be a generic invocation of one of $f_5, \ldots, f_{10}$. First, we need to retrieve the numerical IDs associated to the labels in p (if any). Then, we select an ordering that: 1) allows us to retrieve the answers for p with a range scan; 2) sorts the answers in a way that complies with ω. The orderings that satisfy 1) are

$$\Omega_p = \{\omega' \mid \omega' \in \mathcal{R} \wedge \mathsf{isprefix}(\mathsf{bound}(p), \omega') = true\} \quad (1)$$

An ordering $\omega' \in \Omega_p$ which also satisfies 2) is one for which $\omega' - \mathsf{bound}(p) = \omega - \mathsf{bound}(p)$.

Example 1. *Consider the execution of $\mathsf{edg}_{\mathsf{srd}}(G, p)$ where $p = (X, Y, a)$. In this case, $\mathsf{bound}(p) = \mathsf{d}$, $\Omega_p = \{\mathtt{drs}, \mathtt{dsr}\}$ and $\omega' = \mathtt{dsr}$.*

The selected ω' is associated to one byte stream. If p contains one or more constants, then we can query NM to retrieve the appropriate binary table from that binary stream and (range-)scan it to retrieve the answers of p. In contrast, if p only contains variables, the results can be obtained by scanning all tables in the byte stream. Note that the cost of retrieving the IDs for the labels in p is $O(log|L|)$ since we use B+Trees for the dictionary. This is an operation that is applied any time the input contains a graph pattern. If we ignore this cost and look at the remaining computation, then we can make the following observation.

Proposition 2. *Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of $\mathsf{edg}_\omega(G, p)$ is $O(|E|))$ if p only contains variables, $O(log(|L|) + |E|)$ otherwise.*

Primitives $\mathbf{f_{11}}, \ldots, \mathbf{f_{16}}$ (grp_*). Let $\mathsf{grp}_\omega(G, p)$ be a general call to one of these primitives. Note that in this case $\omega \in \mathcal{R}'$, i.e., is a partial ordering. These functions can be implemented by invoking $f_5, \ldots, f_{10}$ and then return an aggregated version. Thus, they have the same cost as the previous ones.

However, there are special cases where the computation is quicker, as shown in the next example.

Example 2. *Consider a call to $\mathsf{grp}_\mathsf{s}(G, p)$ where $p = \langle a, X, Y\rangle$. In this case, we can query NM with a and return at most one tuple with the cardinality stored in M_a, which has a cost of $O(log(|L|))$.*

If ω has length two or p contains a repeated variable, then we also need to access one or more binary tables, similarly as before.

Proposition 3. *Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of $\mathsf{grp}_\omega(G, p)$ ranges between $O(log(|L|))$ and $O(log(|L|) + |E|)$ depending on p and ω.*

Primitive $\mathbf{f_{17}}$ (count). This primitive returns the cardinality of the output of $f_5, \ldots, f_{16}$. Therefore, it can be simply implemented by iterating over the results returned by these functions. However, there are cases when we can avoid this iteration. Some of such cases are the ones described below:

- **C1** The input is $\text{edg}_\omega(G, p)$ and p contains no constant nor repeated variables. In this case the output is $|E|$.
- **C2** The input is $\text{edg}_\omega(G, p)$ and p contains only one constant c and no repeated variables. Then, the cardinality is stored in M_c.
- **C3** The input is $\text{grp}_\omega(G, p)$, there is an ordering $\omega' \in \Omega_p$ such that $\text{isprefix}(\omega, \omega') = true$, and p contains at most one constant and no repeated variables. Then, the output can be computed either by consulting NM or the metadata of one byte stream.

Otherwise, we also need to access one binary table to compute the results, which, in the worst case, takes $O(|E|)$.

Proposition 4. *Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of executing* count$(\cdot)$ *ranges between $O(log(|L|))$ and $O(log(|L|) + |E|)$.*

Primitives $\mathbf{f_{18}}, \ldots, \mathbf{f_{23}}$ (pos_*). In order to efficiently support these primitives, we need to provide a fast random access to the edges. Given a generic $\text{pos}_\omega(G, p, i)$, we distinguish four cases:

- **C4** If p contains repeated variables, then we iterate over the results of $\text{edg}_\omega(G, p)$ and return the i^{th} edge;
- **C5** If p contains only one constant, then the search space is restricted to a single binary table. In this case, the computation depends on how the content of the table is serialized on the byte stream. If it allows random access to the rows, then the cost reduces to $O(log(|L|))$, i.e., query NM. Otherwise we also need to iterate through the table and count until the i^{th} row;
- **C6** If p contains more than one constant, then we need to perform either a linear or binary search through the table to identify the portion of the table with the answers and retrieve the i^{th} row;
- **C7** Finally, if p does not contain any constants or repeated variables, then we must consider all edges stored in one byte stream. In this case, we first search for the binary table that contains the i^{th} edge. This operation requires a scan of the metadata associated to the byte stream, which can take up to $O(|L|)$. Then, the complexity depends on whether the physical storage of the table allows a random access, as in **C5** and **C6**. If it does not, then the worst-case in terms of complexity is $O(|L| + |E|)$. Note that in this case, simply going through all edges would be faster ($O(|E|)$). However, in practice tables have more than one row so we can advance more quickly despite the higher worst-case complexity.

Proposition 5. *Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of executing $\text{pos}_\omega(G, p, i)$ ranges between $O(log(|L|))$ and $O(|L| + |E|)$.*

4.3 Bulk Loading and Updates

Bulk Loading. Loading a large KG can be a lengthy process, especially if the resources are constrained. In Trident, we developed a loading routine which exploits the multi-core architecture and maximizes the (limited) I/O bandwidth.

The main operations are shown in Figure 2. Our implementation can receive the input KG in multiple formats. Currently, we considered the N-Triples format (popular in the Semantic Web) and the SNAP format [44] (used for generic graphs). The first operation is

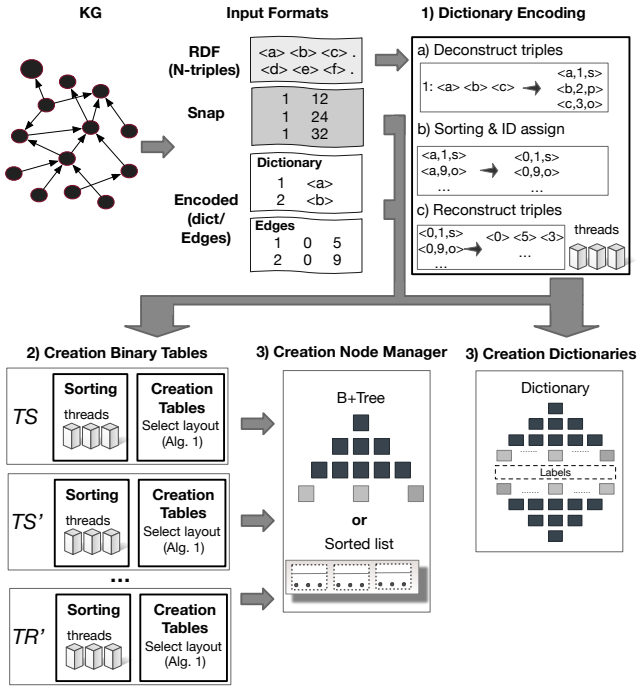


Figure 2: Bulk loading in Trident

encoding the graph, i.e., assigning unique IDs to the entities and relation labels. For this task, we adapted the MapReduce technique presented at [82] to work in a multi-core environment. This technique first deconstructs the triples, then assigns unique IDs to all the terms, and finally reconstruct the triples. If the graph is already encoded, then our procedure skips the encoding and proceeds to the second operation of the loading, the creation of the database.

The creation of the binary tables requires that the triples are presorted according to a given ordering. We use a disk-based parallel merge sort algorithm for this purpose. The tables are serialized one-by-one selecting the most efficient layout for each of them. After all the tables are created, the loading procedure will create the NM and the B+Trees with the dictionaries. The encoding and sorting procedures are parallelized using threads, which might need to communicate with the secondary storage. Modern architectures can have >64 cores, but such a number of threads can easily saturate the disk bandwidth and cause serious slowdowns. To avoid this problem, we have two types of threads: Processing threads, which perform computation like sorting, and I/O threads, which only read and write from disk. In this way, we can control the maximum number of concurrent accesses to the disks.

Updates. To avoid a complete re-loading of the entire KG after each change, our implementation supports incremental updates. Our procedure is built following the well-known advice by Jim Gray [27] that discourages in-place updates, and it is inspired by the idea of differential indexing [55], which proposes to create additional indices and perform a lazy merging with the main database when the number of indices becomes too high.

Our procedure first encodes the update, which can be either an addition or removal, and then stores it in a smaller "delta" database with its own NM and byte streams. Multiple updates will be stored in multiple databases, which are timestamped to remember the order of updates. Also, updates create an extra dictionary if they introduce new terms. Whenever the primitives are executed, the content of the updates is combined with the main KG so that the execution returns an updated view of the graph.

In contrast to differential indexing, our merging does not copy the updates in the main database, but only groups them in two updates, one for the additions and one for the removals. This is to avoid the process of rebuilding binary tables with possibly different layouts. If the size of the merged updates becomes too large, then we proceed with a full reload of the entire database.

5 ADAPTIVE STORAGE LAYOUT

The binary tables can be serialized in different ways. For instance, we can store them row-by-row or column-by-column. Using a single serialization strategy for the entire KG is inefficient because the tables can be very different from each other, so one strategy may be efficient with one table but inefficient with another. Our approach addresses this inefficiency by choosing the best serialization strategy for each table depending on its size and content.

For example, consider two tables T_1 and T_2. Table T_1 contains all the edges with label "isA", while T_2 contains all the edges with label "isbnValue". These two tables are not only different in terms of sizes, but also in the number of duplicated values. In fact, the second column of T_1 is likely to contain many more duplicate values than the second column of T_2 because there are (typically) many more instances than classes while "isbnValue" is a functional property, which means that every entity in the first column is associated with a unique ISBN code. In this case, it makes sense to serialize T_1 in a column-by-column fashion so that we can apply run-length-encoding (RLE) [1], a well-known compression scheme of repeated values, to save space when storing the second column. This type of compression would be ineffective with T_2 since there each value appears only once. Therefore, T_2 can be stored row-by-row.

In our approach, we consider three different serialization strategies, which we call *serialization layouts* (or simply layouts) and employ an ad-hoc procedure to select, for each binary table, the best layout among these three.

5.1 Serialization Layouts

We refer to the three layouts that we consider as *row*, *column*, and *cluster* layouts respectively. The first layout stores the content row-by-row, the second column-by-column, while the third uses an intermediate representation.

Row layout. Let $T = \langle\langle t'_1, t''_1\rangle, \ldots, \langle t'_n, t''_n\rangle\rangle$ be a binary table that contains n sorted pairs of elements. With this layout, the pairs are stored one after the other. In terms of space consumption, this layout is optimal if the two columns do not contain any duplicated value. Moreover, if each row takes a fixed number of bytes, then it is possible to perform binary search or perform a random access to a subset of rows. The disadvantage is that with this layout all values are explicitly written on the stream while the other layouts allow us to compress duplicate values.

Algorithm 1: selectlayout(T)

```
1  U := {u | ⟨u, v⟩ ∈ T}
2  if |T| ≤ τ and |U| ≤ φ then
3      m1 := 0, m2 := 0, m3 := 0
4      foreach u ∈ U do
5          Z := {v | ⟨u, v⟩ ∈ T}
6          if u > m1 then m1 := u
7          if |Z| > m3 then m3 := |Z|
8          foreach z ∈ Z do
9              if z > m2 then m2 := z
10     t_c := |U| * (sizeof(m1) + sizeof(m3)) + |T| * sizeof(m2)
11     t_r := |T| * (sizeof(m1) + sizeof(m2))
12     if t_r ≤ t_c then
13         return ⟨ROW, sizeof(m1), sizeof(m2), 0⟩
14     else return ⟨CLUSTER, sizeof(m1), sizeof(m2), sizeof(m3)⟩
15 else return ⟨COLUMN, 5, 5, 0⟩
```

Column layout. With this layout, the elements in T are serialized as $\langle t'_1, \ldots, t'_n\rangle, \langle t''_1, \ldots, t''_n\rangle$. The space consumption required by this layout is equal to the previous one but with the difference that here we can use RLE to reduce the space of $\langle t'_1, \ldots, t'_n\rangle$. In fact, if $t'_1 = t'_2 = \ldots = t'_n$, then we can simply write $t'_1 \times n$. Also this layout allows binary search and a random access to the table. However, it is slightly less efficient than the row layout for full scans because here one row is not stored at contiguous locations, and the system needs to "jump" between columns in order to return the entire pair. On the other hand, this layout is more suitable than the row layout for aggregate reads (required, for instance, for executing *grp* primitives) because in this case we only need to read the content of one column which is stored at contiguous locations.

Cluster layout. Let $g_t = \langle\langle t, t''_k\rangle, \ldots, \langle t, t''_l\rangle\rangle$ be the longest subsequence of pairs in T which share the first term t. With this layout, all groups are first ordered in the sequence $\langle g_{t_1}, \ldots, g_{t_i}, g_{t_{i+1}}, \ldots, g_{t_m}\rangle$ such that $t_i \leq t_{i+1}$ for all $1 \leq i < m$. Then, they are serialized one-by-one. Each group g_t is serialized by first writing t, then $|g_t|$, and finally the list $t''_k, \ldots, t''_l$. This layout needs less space than the row layout if the groups contain multiple elements. Otherwise, it uses more space because it also stores the size of the groups, and this takes an extra $\lceil log_2 n\rceil$ bits. Another disadvantage is that with this layout binary search is only possible within one group.

5.2 Dynamic Layout Selection

The procedure for selecting the best layout for each table is reported in Algorithm 1. Its goal is to select the layout which leads to the best compression without excessively compromising the performance. In our implementation, Algorithm 1 is applied by default, but the user can disable it and use one layout for all tables.

The procedure receives as input a binary table T and returns a tuple that specifies the layout that should be chosen. It proceeds as follows. First, it makes a distinction between tables that have less than τ rows (default value of τ is 1M) and contain less than ϕ unique elements in the first column from tables that do not (line 2). We make this distinction because 1) if the number of rows is too high then searching for the most optimal layout becomes expensive and 2) if the number of unique pairs is too high, then

the cluster layout should not be used due to the lack of support of binary search. With small tables, this is not a problem because it is well known that in these cases linear search is faster than binary search due to a better usage of the CPU's cache memory. The value for ϕ is automatically determined with a small routine that performs some micro-benchmarks to identify the threshold after which binary search becomes faster. In our experiments, this value ranged between 16 and 64 elements.

If the table satisfies the condition of line 2, then the algorithm selects either the ROW or the CLUSTER layout. The COLUMN layout is not considered because its main benefit against the other two is a better compression (e.g., RLE) but this is anyway limited if the table is small. The procedure scans the table and keeps track of the largest numbers and groups used in the table (m_1, m_2, m_3). Then, the function invokes the subroutine sizeof$(\cdot)$ to retrieve the number of bytes needed to store these numbers. It uses this information to compute the total number of bytes that would be needed to store the table with the ROW and CLUSTER layout respectively (variables t_r and t_c). Then, it selects the layout that leads to maximum compression.

If the condition in line 2 fails, then either the ROW or the COLUMN layout can be selected. An exact computation would be too expensive given the size of the table. Therefore, we always choose COLUMN since the other one cannot be compressed with RLE.

Next to choosing the best layout, Algorithm 1 also returns the maximum number of bytes needed to store the values in the two fields in the table (m_1 and m_2) and (optionally) also for storing the cluster size (m_3, this last value is only needed for CLUSTER). The reason for doing so is that it would be wasteful to use four- or eight-byte integers to store small IDs. In the worst case, we assume that all IDs in both fields can be stored with five bytes, which means it can store up to $2^{40} - 1$ terms. We decided to use byte-wise compression rather than bit-wise compression because the latter does not appear to be worthwhile [55]. Note that more complex compression schemes could also be used (e.g., VByte [86]) but this should be seen as future work.

The tuple returned by selectlayout contains the information necessary to properly read the content of the table from the byte stream. The first field is the chosen layout while the other fields are the number of bytes that should be used to store the entries of the table. We store this tuple both in NM (in one of the m_* fields) and at the beginning of the byte stream.

5.3 Table Pruning

With Algorithm 1, the system adapts to the KG while storing a single table. We discuss two other forms of compression that consider multiple tables and decide whether some tables should be skipped or stored in aggregated form.

On-the-fly reconstruction (OFR). Every table in one stream T_x maps to another table in T'_x where the first column is swapped with the second column. If the tables are sufficiently small, one of them can be re-constructed on-the-fly from the other whenever needed. While this operation introduces some computational overhead, the saving in terms of space may justify it. Furthermore, the overhead can be limited to the first access by serializing the table on disk after the first re-construction.

	Type	#Edges	#Nodes		Type	#Edges	#Nodes
LUBM	KG	Var.	Var.	YAGO2S	KG	76M	37M
DBPedia	KG	1B	233M	Google	Dir.	5.1M	875k
Wikidata	KG	1.1B	299M	Twitter	Dir.	1.7M	81k
Uniprot	KG	168M	177M	Astro	Undi.	198k	18k
BTC2012	KG	1B	367M				

Table 2: Details about the used datasets

We refer to this strategy as *on-the-fly reconstruction (OFR)*. If the user selects it at loading time, the system will not store any binary table in T'_x which has less than η rows, η being a value passed by the user (default value is 20, determined after microbenchmarking).

Aggregate Indexing. Finally, we can construct aggregate indices to further reduce the storage space. The usage of aggregate indices is not novel for KG storage [85]. Here, we limit their usage to the tables in T'_r if they lead to a storage space reduction.

To illustrate the main idea, consider a generic table t that contains the set of tuples $F'_r(isA)$. This table stores all the $\langle object, subject \rangle$ pairs of the triples with the predicate *isA*. Since there are typically many more instances than classes, the first column of t (the classes) will contain many duplicate values. If we range-partition t with the first field, then we can identify a copy of the values in the second field of t in the partitions of tables in T'_d where the first term is *isA*. With this technique, we avoid storing the same sequence of values twice but instead store a pointer to the partition in the other table.

6 EVALUATION

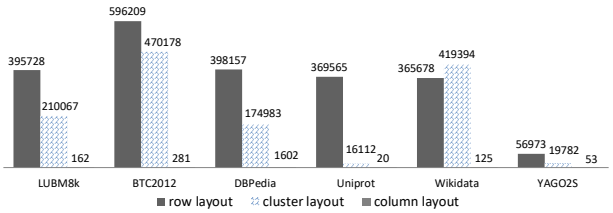
Trident is developed in C++, is freely available, and works under Windows, Linux, MacOS. Trident is also released in the form of a Docker image. The user can interact via command line, web interface, or HTTP requests according to the SPARQL standard.

Integration with other systems. Since our system offers low-level primitives, we integrated it with the following other engines with simple wrappers to evaluate our engine in multiple scenarios:

- **RDF3X [55].** RDF3X is one of the fastest and most well-known SPARQL engines. We replaced its storage layer with ours so that we can reuse its SPARQL operators and query optimizations.
- **SNAP [45].** Stanford Network Analysis Platform (SNAP) is a high-performance open-source library to execute over 100 different graph algorithms. As with RDF3X, we removed the SNAP storage layer and added an interface to our own engine.
- **VLog [81].** VLog is one of most scalable datalog reasoners. We implemented an interface allowing VLog to reason using our system as underlying database.

We also implemented a native procedure to answer basic graph patterns (BGPs) that applies greedy query optimization based on cardinalities, and uses either merge or index loop joins.

Testbed. We used a Linux machine (kernel 3.10, GCC 6.4, page size 4k) with dual Intel E5-2630v3 eight-core CPUs of 2.4 GHz, 64 GB of memory and two 4TB SATA hard disks in RAID-0 mode. The commercial value is well below $5K. We compared against RDF3X and SNAP with their native storages, TripleBit [89], a in-memory state-of-the-art RDF database (in contrast to RDF3X which uses



(a) Number of tables (in 1k) of each type with various KGs

	Graph triple patterns				
	0	1	2	3	4
Default	1.28s	0.07s	**0.15μs**	**0.14μs**	0.18μs
With OFR	1.76s	0.07s	0.38μs	0.38μs	0.30μs
With AGGR	1.30s	0.07s	0.16μs	**0.14μs**	0.19μs
Only ROW	1.25s	0.07s	0.16μs	0.16μs	**0.15μs**
Only COLUMN	1.82s	0.07s	0.22$s\mu$s	0.18μs	0.27μs
RDF3X	**0.70s**	**0.06s**	22.84μs	26.53μs	18.55μs

(b) Median runtimes (best ones are in bold)

Default	With OFR	With AGGR	With OFR
3.9GB	2.7GB	3.4GB	2.5GB

RDF3X: 5.1GB TripleBit: 3.3GB SYSTEM_A: 6.3GB

(c) Size of the database with Trident with/without optimizations

Figure 3: Statistics using various layouts/configurations and runtimes of triple pattern lookups

disks), and SYSTEM_A, a widely used commercial SPARQL engine[1]. As inputs, we considered a selection of real-world and artifical KGs, and other non-KG graphs from SNAP [44] (see Table 2 for statistics).

- **KGs.** LUBM [30], a well-known artificial benchmark that creates KGs of arbitrary sizes. The KG is in the domain of universities and each university contributes ca. 100k new triples. Henceforth, we write *LUBMX* to indicate a KG with X universities, e.g. LUBM10 contains 1M triples; DBPedia [13], YAGO2S [75] and Wikidata [84], three widely used KGs with encyclopedic knowledge; Uniprot [67], a KG that contains biomedical knowledge, and BTC2012 [37], a collection of crawled interlinked KGs.
- **Other graphs.** We considered the graphs: Google, a Web graph from Google, Twitter, which contains a social circle, and Astro, a collaboration network in Physics.

In the following, Trident was configured to use the B+Tree for NM and table pruning was disabled, unless otherwise stated.

6.1 Lookups

During the loading procedure, Trident applies Algorithm 1 to determine the best layout for each table. Figure 3a shows the number of tables of each type for the KGs. The vast majority of tables is stored either with the ROW or CLUSTER layout. Only a few tables are stored with the COLUMN layout. These are mostly the ones in the TR and TR′ byte streams. It is interesting to note that the number of tables varies differently among different KGs. For instance, the number of row tables is twice the number of cluster tables with LUBM. In contrast, with Wikidata there are more cluster tables than row ones. These differences show to what extent Trident adapted its physical storage to the structure of the KG.

One key operation of our system is to retrieve answers of simple triple patterns. First, we generated all possible triple patterns that return non-empty answers from YAGO2S. We considered five types of patterns. Patterns of type 0 are full scans; of type 2 contain one constants and two variables (e.g., X, *type*, Y), while of type 4 contain two constants and one variable (e.g., X, *type*, *person*). These patterns are answered with edg_*. Patterns of types 1 request an aggregated version of a full scan (e.g., retrieve all subjects) while patterns of type 3 request an aggregation where the pattern contains one constant (e.g., return all objects of the predicate *type*). These two patterns are answered with grp_*.

[1]We hide the real name as it is a commercial product, as usual in database research.

We used the primitives to retrieve the answers for these patterns with various configurations of our system, and compared against RDF3X, which was the system with the fastest runtimes. The median warm runtimes of all executions are reported in Figure 3b.

The row "Default" reports the results with the adaptive storage selected by Algorithm 1 but without table pruning. The rows "With OFR" and "With AGGR" use Algorithm 1 and the two techniques for table pruning discussed in Section 5.3 respectively. The rows "Only ROW (COLUMN)" use only the ROW and COLUMN layouts (the CLUSTER is not competitive alone due to the lack of binary search). From the table, we see that if the two pruning strategies are enabled, then the runtimes increase, especially with OFR. This was expected since these two techniques trade speed for space. Their benefit is that they reduce the size of the database, as shown in Figure 3c. In particular, OFR is very effective, and they can reduce the size by 35%. Therefore, they should only be used if space is critical. The ROW layout returns competitive performance if used alone but then the database size is about 9% larger due to the suboptimal compression. Figure 3c also reports the size of the databases with the other systems as reference. Note that the reported numbers for Trident do not include the size of the dictionary (764MB). This size should be added to the reported numbers for a fair comparison with the other systems' databases.

A comparison against RDF3X shows that the latter is faster with full scans (patterns of type 0) because our approach has to visit more tables stored with different configurations. However, our approach has comparable performance with the second pattern type and performs significantly better when the scan is limited to a single table, with, in the best case, improvements of more than two orders of magnitude (pattern 3). Note that in contexts like SPARQL query answering, patterns that contain at least one constant are much more frequent than full scans (e.g., see Table 2 of [68]).

6.2 SPARQL

Figure 4a shows the average of five warm runtimes with our system and with other state-of-the-art engines. For LUBM8k, DBPedia, Uniprot, and BTC2012, we considered some of the queries used to evaluate TripleBit [89]. For Wikidata, we designed example queries of various complexity looking at published examples. All queries are available at [80]. Unfortunately, we could not load Wikidata and BTC2012 with SYSTEM_A due to errors during the loading phase.

We can make a few observations from the obtained results. First, a direct comparison against TripleBit is problematic because

	Q.	N. Query Answers	Avg warm runtime (ms.) TN	TR	R3X	TripleBit	SYSTEM_A
LUBM8k	1	10	**0.107**	0.164	1.331	11.76	0.39
	2	10	**0.181**	1,067	3.528	x	0.33
	3	0	806,194	2,387	701,738	**0.04**	9,365
	4	2,528	611,083	19,435	709,541	20,298	**9,244**
	5	351,919	**4,704**	14,452	22,468	12,212	150,407
Uniprot	1	853	**1.245**	8.676	15.058	2.347	17.145
	2	32	**0.168**	52.000	4.144	0.545	1.121
	3	10,715,646	17,729	7,318	64,923	**6,532**	96,063
	4	5,564	**17.025**	438.864	50.898	211.71	58.498
	5	0	2.759	298.881	2.489	x	**1.226**
DBPEDIA	1	449	**0.632**	80.511	21.157	3.14	25.86
	2	600	**0.075**	0.118	3.533	0.58	4.47
	3	270	**0.436**	7.761	9.356	0.92	3.22
	4	68	**0.371**	124.391	9.600	1.42	4.32
	5	1,643	**5.158**	24.207	31.842	8.89	47.46
BTC2012	1	0	0.069	**0.050**	4.903	0.99	N.A.
	2	1	**0.197**	24.911	6.279	0.55	N.A.
	3	1	**0.257**	116.572	18.896	1.80	N.A.
	4	664	**24.693**	1,049	290.655	75.98	N.A.
	5	5,996	**5.493**	9,410	528.668	18.95	N.A.
WIKIDATA	1	43	**0.183**	0.386	2.278	0.505	N.A.
	2	55	0.066	**0.063**	1.081	0.107	N.A.
	3	1,583	**4.095**	140.315	52.731	57.054	N.A.
	4	682	24.110	27.627	179.36	**15.219**	N.A.
	5	1,975,090	817.614	**728.641**	6,399	1,533	N.A.

(a) Runtime SPARQL queries. "TN" is the runtime of our SPARQL engine while "TR" uses our storage and the RDF3X SPARQL engine. Red background means that TripleBit returned wrong answers ('x' means it crashed); "N.A." means that there was a failure at loading time

Task	ASTRO Snap	ASTRO Ours	GOOGLE Snap	GOOGLE Ours	TWITTER Snap	TWITTER Ours
HITS	431	**89**	9252	**3557**	2399	**588**
BFS	81993	**62241**	**1604037**	1709823	**215704**	243740
Triangles	69	**31**	1353	**526**	607	**105**
Random Walks	**25**	34	**30**	41	**26**	32
MaxWCC	22	**15**	594	**351**	132	**65**
MaxSCC	47	**29**	1177	**712**	228	**148**
Diameter	11767	**5233**	**168211**	243669	56132	**42581**
PageRank	515	**319**	14616	**7771**	4482	**1738**
ClustCoef	**375**	417	8519	**7114**	5886	**4178**
mod	7	**6**	**5**	23	**8**	13

(b) Runtime of graph analytics algorithms (ms)

Datalog reasoning using LUBM1k (130M triples)

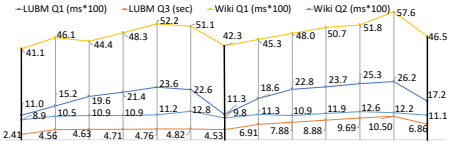
Ruleset from [52]	VLog+Ours	VLog
LUBM-L	**17.3s**	25.6s
LUBM-LE	**31m**	34m

Runtime training 10 epochs with TransE and YAGO

Params: bathsize=100,learningrate=0.001,dims=50,adagrad,margin=1

Ours: **8.6s** OpenKE [33]: 18.72s

(c) Runtime of reasoning and learning



(d) Warm runtimes on Wikidata and LUBM8k after adding/removing 1M triples

Universities (# facts)	Q1 (ms)	Q2 (ms)	Q3
10k (1.3B)	0.05	0.09	25m
20k (2.6B)	0.05	0.09	52m
40k (5B)	0.05	0.09	1h50m
80k (10B)	0.05	0.09	3h52m
160k (21B)	0.05	0.09	>8h
800k (100B)	0.05	0.09	>8h
	Q4	**Q5**	
10k (1.3B)	11m	6s	
20k (2.6B)	41m	12s	
40k (5B)	1h42s	25s	
80k (10B)	3h1m	56s	
160k (21B)	6h49m	1m51s	
800k (100B)	>8h	12m	

(e) Runtime LUBM Q1-Q5 and KGs which size ranges between 1-100B triples

Figure 4: Performance of our storage engine on several tasks, with larger KGs and updates

sometimes `TripleBit` crashed or returned wrong results (checked after manual inspection). Looking at the other systems, we observe that our system is faster 20 out of 25 times. Furthermore, we observe that TN (our native SPARQL engine) is faster than TR (our storage+RDF3X) mostly with selective queries that require fewer joins (e.g., LUBM Q1,Q2, DBPedia Q1, Q2, Q3, Wikidata Q1, Q3). The reason is that `RDF3X` uses a sophisticated query optimizer that considers more plans in a bottom-up fashion. This query optimization procedure is costly and pays off only with more complex queries.

6.3 Graph Analytics, Reasoning, Learning

Graph analytics. Algorithms for graph analytics are used for path analysis (e.g., find the shortest paths), community analysis (e.g., triangle counting), or to compute centrality metrics (e.g., PageRank). They use frequently the primitives pos_* and *count* to traverse the graph or to obtain the nodes' degree. For these experiments, we used the sorted list as `NODEMGR` since these algorithms are node-centric.

We selected ten well-known algorithms: *HITS* and *PageRank* compute centrality metrics; *Breadth First Search (BFS)* performs a search; *MOD* computes the modularity of the network, which is used for community detection; *Triangle Counting* counts all triangles; *Random Walks* extracts random paths; *MaxWCC* and *MaxSCC* compute the largest weak and strong connected components respectively; *Diameter* computes the diameter of the graph while *ClustCoeff* computes the clustering coefficient.

We executed these algorithms using the original SNAP library and in combination with our engine. Note that the implementation of the algorithms is the same; only the storage changes. Figure 4b reports the runtimes. From it, we see that our engine is faster in most cases. It is only with random walks that our approach is slower. From these results, we conclude that also with this type of computation our approach leads to competitive runtimes.

Reasoning and Learning. We also tested the performance of our system for rule-based reasoning. In this task, rules are used to materialize all possible derivations from the KG. First, we computed reasoning considering Trident and VLog, using LUBM and two popular rulesets (LUBM-L and LUBM-LE) [52, 81]. Then, we repeated the process with the native storage of VLog. The runtime, reported in Figure 4c, shows that our engine leads to an improvement of the performance (48% faster in the best case).

Finally, we considered statistical relational learning as another class of problems that could benefit from our engine. We implemented TransE [15], one of the most popular techniques of this kind, on top of Trident and compared the runtime of training vs. the one produced by OpenKE [33], a state-of-the-art library. Figure 4c reports the runtime to train a model using as input a subset of YAGO which was used in other works [61]. The results indicate competitive performance also in this case.

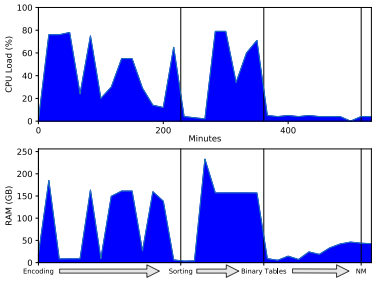
6.4 Scalability, updates, and bulk loading

We executed the five LUBM queries using our native SPARQL procedure on KGs of different sizes (between 1B-100B triples). We used another machine with 256GB of RAM for these experiments (which also costs <$5K) due to lack of disk space. The warm runtimes are shown in Figure 4e. The runtime of the first two queries remains constant. This was expected since their selectivity does not decrease as the size of the KG increases. In contrast, the runtime of the other queries increases as the KG becomes larger.

Figure 4d shows the runtime of four SPARQL queries after we added five sets of about 1M new triples to the KG, merged them

Op	Wikidata	LUBM8K
ADD	308s	175s
ADD	386s	230s
ADD	404s	242s
ADD	477s	261s
ADD	431s	260s
Merge	200s	114s
DEL	399s	222s
DEL	465s	278s
DEL	501s	319s
DEL	531s	342s
DEL	566s	369s
Merge	291s	181s

(a) Runtime of updates



(b) CPU/RAM usage loading LUBM80k

System	Runtime	System	Runtime	System	Runtime
Ours (seq)	20min	RDF3X (seq)	24min	TripleBit (par)	9min
Ours (par)	6min	SYSTEM_A (par)	1h9min		

(c) Loading runtime of LUBM1k (130M triples)

Figure 5: Loading and update runtimes

into single updates, removed five other sets of 1M triples, and then merged again. We selected the queries so that the content of the updates is considered. We observe that the runtime increases (because more deltas are considered) and that it drops after they are merged in a single update. Figure 5a reports the runtime to process five additions of ca. 1M novel triples, one merge, five removals of ca. 1M existing triples, and another merge. As we can see, with both datasets the runtime is much smaller than re-creating the database from scratch (>1h). The runtime with LUBM8k is faster than with Wikidata because the updates with the latter KG contained 4X more new entities.

In Figure 5b, we show the trace of the resource consumption during the loading of LUBM80k (10B triples). We plot the CPU (100% means all physical cores are used) and RAM usage. From it, we see that most of the runtime is taken to dictionary encoding, sorting the edges, and to create the binary tables.

In general, Trident has competitive loading times. Figure 5c shows the loading time of ours and other systems on LUBM1k. With larger KGs, RDF3X becomes significantly slower than ours (e.g., it takes ca. 7 hours to load LUBM8k on our smaller machine while Trident needs 1 hour and 18 minutes) due to lack of parallelism. TripleBit is an in-memory database and thus it cannot scale to some of our largest inputs. In some of our largest experiments, Trident could load LUBM400k (50B triples) in about 48 hours which is a size that other systems cannot handle. If the graph is already encoded, then loading is faster. We loaded the Hyperlink Graph [50], a graph with 128B edges, in about 13 hours (with the larger machine) and the database required 1.4TB of space.

7 RELATED WORK

In this section, we describe the most relevant works to our problem. For a broader introduction to graph and RDF processing, we redirect to existing surveys [3, 22, 49, 51, 60, 69, 87]. Current approaches can be classified either as *native* (i.e., designed for this task) or *non-native* (adapt pre-existing technology). Native engines have better performance [16], but less functionalities [16, 21]. Our approach belongs to the first category.

Research on native systems has focused on advanced indexing structures. The most popular approach is to extensively materialize a dedicated index for each permutation. This was initially proposed by YARS [38], and further explored in RDF3X [10–12, 24, 55]. Also Hexastore [85] proposes a six-way permutation-based indexing, but implemented it using hierarchical in-memory Java hash maps. Instead, we use on-disk data structures and therefore can scale to larger inputs. Recently, other types of indices, based on 2D or 3D bit matrices [8, 89], hash-maps [52], or data structures used for graph matching approaches [42, 91] have been proposed. If compared with these works, our approach uses a novel layout of data structures and uses multiple layouts to store the subgraphs.

Non-native approaches offload the indexing to external engines (mostly DBMS). Here, the challenge is to find efficient partitioning/replication criteria to exploit the multi-table nature of relational engines. Existing partitioning criteria group the triples either by predicates [2, 35, 46, 48, 54, 64, 65], clusters of predicates [19], or by using other entity-based splitting criteria [16]. The various partitioning schemes are designed to create few tables to meet the constraints of relational engines [73]. Our approach differs because we group the edges at a much higher granularity generating a number of binary tables that is too large for such engines.

Some popular commercial systems for graph processing are Virtuoso [59], BlazeGraph [76], Titan [20], Neo4J [53] Sparksee [74], and InfiniteGraph [58]. We compared Trident against such a leading commercial system and observed that ours has very competitive performance; other comparisons are presented in [5, 73]. In general, a direct comparison is challenging because these systems provide end-to-end solutions tailored for specific tasks while we offer general-purpose low-level APIs.

Finally, many works have focused on distributed graph processing [4, 6, 9, 31, 34, 35, 43, 62, 70, 90]. We do not view these approaches as competitors since they operate on different hardware architectures. Instead, we view ours as a potential complement that can be employed by them to speed up a distributed processing of very large graphs.

8 CONCLUSION

In this paper, we proposed a novel centralized architecture for the low-level storage of very large KGs which provides both node- and edge-centric access to the KG. One of the main novelties of our approach is that it exhaustively decomposes the storage of the KGs in many binary tables, serializing them in multiple byte streams to facilitate inter-table scanning, akin to permutation-based approaches. Another main novelty is that the storage effectively adapts to the KG by choosing a different layout for each table depending on the graph topology. Our empirical evaluation, performed using a prototype called Trident, shows that our approach is competitive in multiple scenarios and that it can load very large graphs without expensive hardware.

Future work is necessary to apply or adapt our architecture for additional scenarios. In particular, we believe that our system can be used to support Triple Pattern Fragments [83], an emerging paradigm to query RDF datasets, and GraphQL [39], a more complex graph query language. Finally, it is also interesting to study whether

the integration of additional compression techniques, like locality-based dictionary encoding [79] or HDT [23], can further improve the runtime and/or reduce the storage space.

Acknowledgments. We would like to thank (in alphabetical order) Peter Boncz, Martin Kersten, Stefan Manegold, and Gerhard Weikum for discussing and providing comments to improve this work. This project was partly funded by the NWO research programme 400.17.605 (VWData) and NWO VENI project 639.021.335. Some experiments presented in this paper were performed on the clusters DAS5 and SCILENS funded by NWO grants.

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of SIGMOD*. ACM, New York, NY, USA, 671–682.

[2] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. 2009. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *The VLDB Journal* 18, 2 (2009), 385–406.

[3] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *PVLDB* 10, 13 (2017), 2049–2060.

[4] I. Abdelaziz, R. Harbi, S. Salihoglu, and P. Kalnis. 2017. Combining vertex-centric graph processing with SPARQL for large-scale RDF data analytics. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3374–3388.

[5] Günes Aluç, M. Tamer Özsu, Khuzaima Daudjee, and Olaf Hartig. 2015. Executing queries over schemaless RDF databases. In *Proceedings of ICDE*. IEEE, Seoul, South Korea, 807–818.

[6] Bernd Amann, Olivier Curé, and Hubert Naacke. 2018. *Distributed SPARQL Query Processing: a Case Study with Apache Spark*. John Wiley & Sons, Ltd, Hoboken, NJ, USA, Chapter 2, 21–55.

[7] Grigoris Antoniou, Sotiris Batsakis, Raghava Mutharaju, Jeff Z. Pan, Guilin Qi, Ilias Tachmazidis, Jacopo Urbani, and Zhangquan Zhou. 2018. A survey of large-scale reasoning on the Web of data. *The Knowledge Engineering Review* 33 (2018), 1–43.

[8] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. 2010. Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *Proceedings of WWW*. ACM, New York, NY, USA, 41–50.

[9] A. Azzam, S. Kirrane, and A. Polleres. 2018. Towards Making Distributed RDF Processing FLINKer. In *2018 4th International Conference on Big Data Innovations and Applications (Innovate-Data)*. IEEE, Los Alamitos, CA, USA, 9–16.

[10] Liu Baolin and Hu Bo. 2007. HPRD: A High Performance RDF Database. In *Proceedings of NPC*. Springer, Cham, Switzerland, 364–374.

[11] David Beckett. 2001. The Design and Implementation of the Redland RDF Application Framework. In *Proceedings of WWW*. ACM, New York, NY, USA, 449–456.

[12] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. 2011. OWLIM: A family of scalable semantic repositories. *Semantic Web* 2, 1 (2011), 33–42.

[13] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia-A crystallization point for the Web of Data. *Web Semantics: science, services and agents on the world wide web* 7, 3 (2009), 154–165.

[14] Roi Blanco, Berkant Barla Cambazoglu, Peter Mika, and Nicolas Torzec. 2013. Entity Recommendations in Web Search. In *Proceedings of ISWC*. Springer, Heidelberg, Germany, 33–48.

[15] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Proceedings of NIPS*. NIPS Proceedings, Lake Tahoe, NV, USA, 2787–2795.

[16] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an Efficient RDF Store over a Relational Database. In *Proceedings of SIGMOD*. ACM, New York, NY, USA, 121–132.

[17] Alison Callahan, José Cruz-Toledo, Peter Ansell, and Michel Dumontier. 2013. Bio2RDF Release 2: Improved Coverage, Interoperability and Provenance of Life Science Linked Data. In *Proceedings of ESWC*. Springer, Heidelberg, Germany, 200–212.

[18] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *PVLDB* 8, 12 (2015), 1804–1815.

[19] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. 2005. An Efficient SQL-Based RDF Querying Scheme. In *Proceedings of VLDB*. VLDB Endowment, Trondheim, Norway, 1216–1227.

[20] DATASTAX, Inc. 2019. Titan: Distributed Graph Database. http://titan.thinkaurelius.com/

[21] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *Proceedings of CIDR*. www.cidrdb.org, Asilomar, CA, USA.

[22] David C. Faye, Olivier Curé, and Guillaume Blin. 2011. A survey of RDF storage approaches. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées* 15 (2011), 11–35.

[23] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF representation for publication and exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web* 19 (2013), 22–41.

[24] George H.L. Fletcher and Peter W. Beck. 2009. Scalable Indexing of RDF Graphs for Efficient Join Processing. In *Proceedings of CIKM*. ACM, New York, NY, USA, 1513–1516.

[25] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of OSDI*. USENIX, Hollywood, CA, USA, 17–30.

[26] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of OSDI*. USENIX, Broomfield, CO, USA, 599–613.

[27] Jim Gray. 1981. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of VLDB*. VLDB Endowment, Cannes, France, 144–154.

[28] Mark Greaves and Peter Mika. 2008. Semantic Web and Web 2.0. *Web Semantics: Science, Services and Agents on the World Wide Web* 6, 1 (2008), 1–3.

[29] R. Guha, Rob McCool, and Eric Miller. 2003. Semantic Search. In *Proceedings of WWW*. ACM, New York, NY, USA, 700–709.

[30] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3, 2 (2005), 158–182.

[31] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: A Distributed Shared-Nothing RDF Engine Based on Asynchronous Message Passing. In *Proceedings of SIGMOD*. ACM, New York, NY, USA, 289–300.

[32] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. 2014. An Experimental Comparison of Pregel-like Graph Processing Systems. *PVLDB* 7, 12 (2014), 1047–1058.

[33] Xu Han, Shulin Cao, Xin Lv, Yankai Lin, Zhiyuan Liu, Maosong Sun, and Juanzi Li. 2018. OpenKE: An Open Toolkit for Knowledge Embedding. In *Proceedings of EMNLP*. ACL, Brussels, Belgium, 139–144.

[34] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. 2016. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal* 25, 3 (2016), 355–380.

[35] Steve Harris, Nick Lamb, and Nigel Shadbolt. 2009. 4store: The Design and Implementation of a Clustered RDF Store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*. CEUR Workshop Proceedings, Washington, DC, USA, 94–109.

[36] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query

[37] Andreas Harth. 2012. Billion Triples Challenge data set. http://km.aifb.kit.edu/projects/btc-2012/

[38] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. 2007. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *Proceedings of ISWC*. Springer, Heidelberg, Germany, 211–224.

[39] Olaf Hartig and Jorge Pérez. 2018. Semantics and Complexity of GraphQL. In *Proceedings of WWW*. International World Wide Web Conferences Steering Committee, Geneva, Switzerland, 1155–1164.

[40] Patrick Hayes. 2004. RDF Semantics, W3C Recommendation. http://www.w3.org/TR/rdf-mt/

[41] Gjergji Kasneci, Fabian M. Suchanek, Georgiana Ifrim, Maya Ramanath, and Gerhard Weikum. 2008. NAGA: Searching and Ranking Knowledge. In *Proceedings of ICDE*. IEEE, Cancun, Mexico, 953–962.

[42] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming Subgraph Isomorphism for RDF Query Processing. *PVLDB* 8, 11 (2015), 1238–1249.

[43] Kisung Lee and Ling Liu. 2013. Scaling queries over big RDF graphs with semantic hash partitioning. *PVLDB* 6, 14 (2013), 1894–1905.

[44] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[45] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[46] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. 2004. RStar: An RDF Storage and Query System for Enterprise Resource Management. In *Proceedings of CIKM*. ACM, New York, NY, USA, 484–491.

[47] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of SIGMOD*. ACM, New York, NY, USA, 135–146.

[48] Brian McBride. 2001. Jena: Implementing the RDF Model and Syntax Specification. In *Semantic Web Workshop 2001*. CEUR Workshop Proceedings, Hong Kong, 23–28.

[49] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 25.

[50] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The Graph Structure in the Web – Analyzed on Different Aggregation Levels. *The Journal of Web Science* 1, 1 (2015), 33–47.

[51] G. E. Modoni, M. Sacco, and W. Terkaj. 2014. A Survey of RDF Store Solutions. In *Proceedings of ICE*. IEEE, Bergamo, Italy, 1–7.

[52] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. 2014. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *Proceedings of AAAI*. AAAI Press, Québec, Canada, 129–137.

[53] Neo4j, Inc. 2019. Neo4j Graph Platform. https://neo4j.com/

[54] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *Proceedings of ICDE*. IEEE, Hannover, Germany, 984–994.

[55] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19, 1 (2010), 91–113.

[56] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2015. A review of relational machine learning for knowledge graphs. *Proc. IEEE* 104, 1 (2015), 11–33.

[57] Natalya F. Noy, Nigam H. Shah, Patricia L. Whetzel, Benjamin Dai, Michael Dorf, Nicholas Griffith, Clement Jonquet, Daniel L. Rubin, Margaret-Anne Storey, and Christopher G. Chute. 2009. BioPortal: ontologies and integrated data resources at the click of a mouse. *Nucleic acids research* 37, suppl_2 (2009), W170–W173.

[58] Objectivity Inc. 2019. InfiniteGraph. https://www.objectivity.com/products/infinitegraph/

[59] OpenLink Software. 2019. Virtuoso RDF Engine. https://virtuoso.openlinksw.com/

[60] M. Tamer Özsu. 2016. A survey of RDF data management systems. *Frontiers of Computer Science* 10, 3 (2016), 418–432.

[61] Soumajit Pal and Jacopo Urbani. 2017. Enhancing Knowledge Graph Completion By Embedding Correlations. In *Proceedings of CIKM*. ACM, New York, NY, USA, 2247–2250.

[62] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. 2016. Processing SPARQL queries over distributed RDF graphs. *The VLDB Journal* 25, 2 (2016), 243–268.

[63] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. 2015. Ringo: Interactive Graph Analytics on Big-Memory Machines. In *Proceedings of SIGMOD*. ACM, New York, NY, USA, 1105–1110.

[64] Minh-Duc Pham and Peter Boncz. 2016. Exploiting Emergent Schemas to Make RDF Systems More Efficient. In *Proceedings of ISWC*. Springer, Basel, Switzerland, 463–479.

[65] Minh-Duc Pham, Linnea Passing, Orri Erling, and Peter Boncz. 2015. Deriving an Emergent Relational Schema from RDF Data. In *Proceedings of WWW*. International World Wide Web Conferences Steering Committee, Geneva, Switzerland, 864–874.

[66] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. 2014. Scalable Big Graph Processing in MapReduce. In *Proceedings of SIGMOD*. ACM, New York, NY, USA, 827–838.

[67] Nicole Redaschi and UniProt Consortium. 2009. UniProt in RDF: Tackling Data Integration and Distributed Annotation with the Semantic Web. *Nature Precedings* (2009).

[68] Laurens Rietveld and Rinke Hoekstra. 2014. YASGUI: Feeling the Pulse of Linked Data. In *Proceedings of EKAW*. Springer, Basel, Switzerland, 441–452.

[69] Sherif Sakr and Ghazi Al-Naymat. 2010. Relational Processing of RDF Queries: A Survey. *SIGMOD Record* 38, 4 (2010), 23–28.

[70] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF Querying with SPARQL on Spark. *PVLDB* 9, 10 (2016), 804–815.

[71] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of SIGMOD*. ACM, New York, NY, USA, 505–516.

[72] W. Shen, J. Wang, and J. Han. 2015. Entity linking with a knowledge base: issues, techniques, and solutions. *IEEE Transactions on Knowledge and Data Engineering* 27, 2 (2015), 443–460.

[73] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. 2008. Column-Store Support for RDF Data Management: not all swans are white. *PVLDB* 1, 2 (2008), 1553–1563.

[74] Sparsity Technologies. 2019. Sparksee. http://sparsity-technologies.com/

[75] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2008. YAGO: A Large Ontology from Wikipedia and WordNet. *Web Semantics: Science, Services and Agents on the World Wide Web* 6, 3 (2008), 203–217.

[76] Systap. 2019. BlazeGraph. https://blazegraph.com/

[77] Niket Tandon, Gerard de Melo, Fabian M. Suchanek, and Gerhard Weikum. 2014. WebChild: Harvesting and Organizing Commonsense Knowledge from the Web. In *Proceedings of WSDM*. ACM, New York, NY, USA, 523–532.

[78] Alberto Tonon, Michele Catasta, Roman Prokofyev, Gianluca Demartini, Karl Aberer, and Philippe Cudre-Mauroux. 2016. Contextualized ranking of entity types based on knowledge graphs. *Web Semantics: Science, Services and Agents on the World Wide Web* 37 (2016), 170–183.

[79] Jacopo Urbani, Sourav Dutta, Sairam Gurajada, and Gerhard Weikum. 2016. KOGNAC: Efficient Encoding of Large Knowledge Graphs. In *Proceedings of IJCAI*. AAAI Press, New York, NY, USA, 3896–3902.

[80] Jacopo Urbani and Ceriel Jacobs. 2020. Adaptive Low-level Storage of Very Large Knowledge Graphs. arXiv:2001.09078

[81] Jacopo Urbani, Ceriel Jacobs, and Markus Krötzsch. 2016. Column-Oriented Datalog Materialization for Large Knowledge Graphs. In *Proceedings of AAAI*. AAAI Press, Phoenix, AZ, USA, 258–264.

[82] Jacopo Urbani, Jason Maassen, Niels Drost, Frank Seinstra, and Henri Bal. 2013. Scalable RDF data compression with MapReduce. *Concurrency and Computation: Practice and Experience* 25, 1 (2013), 24–39.

[83] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Web Semantics: Science, Services and Agents on the World Wide Web* 37 (2016), 184–206.

[84] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledge base. *Commun. ACM* 57, 10 (2014), 78–85.

[85] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. *PVLDB* 1, 1 (2008), 1008–1019.

[86] Hugh E. Williams and Justin Zobel. 1999. Compressing Integers for Fast File Access. *Comput. J.* 42, 3 (1999), 193–201.

[87] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. 2018. RDF Data Storage and Query Processing Schemes: A Survey. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 84:1–84:36.

[88] Mohamed Yahya, Denilson Barbosa, Klaus Berberich, Qiuyue Wang, and Gerhard Weikum. 2016. Relationship Queries on Extended Knowledge Graphs. In *Proceedings of WSDM*. ACM, New York, NY, USA, 605–614.

[89] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a fast and compact system for large scale RDF data. *PVLDB* 6, 7 (2013), 517–528.

[90] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. *PVLDB* 6, 4 (2013), 265–276.

[91] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. 2014. gStore: a graph-based SPARQL query engine. *The VLDB Journal* 23, 4 (2014), 565–590.